* **Iterative Fibonacci Program**

# Assignment No 1

# Program to display the Fibonacci sequence up to n-th term nterms = int(input("Enter number of terms: "))

# first two terms n1, n2 = 0, 1

count = 0

# check if the number of terms is valid if nterms <= 0:

print("Please enter a positive integer") # if there is only one term, return n1

elif nterms == 1:

print("Fibonacci sequence up to", nterms, ":") print(n1)

# generate fibonacci sequence else:

print("Fibonacci sequence:") while count < nterms:

print(n1)

nth = n1 + n2

# update values n1 = n2

n2 = nth count += 1

## Output :

Enter number of terms: 4

Fibonacci sequence: 0

1

1

2

## Recursive Fibonacci Program

# Recursive function for Fibonacci sequence def fibonacci(n):

if n <= 1: return n

else:

return fibonacci(n-1) + fibonacci(n-2)

n = int(input("Enter number of terms: ")) print("Fibonacci sequence:")

for i in range(n): print(fibonacci(i))

## Output :

Enter number of terms: 4 Fibonacci sequence:

0

1

1

2

# Assignment No. 2

## Huffman Tree Implementation

# Huffman Tree Node class class node:

def init (self, freq, symbol, left=None, right=None): self.freq = freq # Frequency of the symbol self.symbol = symbol # Character symbol

self.left = left # Left child of the node self.right = right # Right child of the node self.huff = '' # Huffman code (0 or 1)

# Function to print the Huffman codes def printNodes(node, val=''):

newVal = val + str(node.huff) if node.left:

printNodes(node.left, newVal) if node.right:

printNodes(node.right, newVal) if not node.left and not node.right:

print(f"{node.symbol} -> {newVal}")

# Characters for Huffman tree chars = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Frequency of characters

freq = [4, 7, 12, 14, 17, 43, 54]

# List containing unused nodes nodes = []

# Converting characters and frequencies into Huffman tree nodes for x in range(len(chars)):

nodes.append(node(freq[x], chars[x]))

while len(nodes) > 1:

# Sort all the nodes in ascending order based on their frequency nodes = sorted(nodes, key=lambda x: x.freq)

# Pick 2 smallest nodes left = nodes[0]

right = nodes[1]

# Assign directional value to these nodes left.huff = 0

right.huff = 1

# Combine the 2 smallest nodes to create new node as their parent newNode = node(left.freq + right.freq, left.symbol + right.symbol, left, right)

# Remove the 2 nodes and add their parent as new node among others nodes.remove(left)

nodes.remove(right)

nodes.append(newNode)

# Huffman Tree is ready! Print the codes printNodes(nodes[0])

## Output :

a -> 0000

b -> 0001

c -> 001

d -> 010

e -> 011

f -> 10

g -> 11

# Assignment No. 3

## Fractional Knapsack Implementation

def fractional\_knapsack(value, weight, capacity):

# index = [0, 1, 2, ..., n - 1] for n items index = list(range(len(value)))

# contains ratios of values to weight

ratio = [v / w for v, w in zip(value, weight)]

# index is sorted according to value-to-weight ratio in decreasing order index.sort(key=lambda i: ratio[i], reverse=True)

max\_value = 0 # Maximum value of items

fractions = [0] \* len(value) # Fractions of each item to be taken

for i in index:

if weight[i] <= capacity:

# Take the whole item fractions[i] = 1

max\_value += value[i] capacity -= weight[i]

else:

# Take fraction of the item fractions[i] = capacity / weight[i]

max\_value += value[i] \* capacity / weight[i] break # The knapsack is full

return max\_value, fractions # Input section

n = int(input('Enter number of items: '))

value = input(f'Enter the values of the {n} item(s) in order: ').split() value = [int(v) for v in value]

weight = input(f'Enter the positive weights of the {n} item(s) in order: ').split() weight = [int(w) for w in weight]

capacity = int(input('Enter maximum weight: '))

# Calculate maximum value and fractions

max\_value, fractions = fractional\_knapsack(value, weight, capacity)

# Output section

print('The maximum value of items that can be carried:', max\_value) print('The fractions in which the items should be taken:', fractions)

## Output :

Enter number of items: 3

Enter the values of the 3 item(s) in order: 24 15 25

Enter the positive weights of the 3 item(s) in order: 15 10 18 Enter maximum weight: 20

The maximum value of items that can be carried: 31.5

The fractions in which the items should be taken: [1, 0.5, 0]

# Assignment No. 4

## N-Queen Problem

# Python program to solve N Queen Problem using backtracking # Size of the chessboard

N = 4

# Function to print the solution def printSolution(board):

for row in board:

print(" ".join(str(cell) for cell in row)) print()

# Utility function to check if a queen can be placed on board[row][col] def isSafe(board, row, col):

# Check this row on the left side for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on the left side i, j = row, col

while i >= 0 and j >= 0: if board[i][j] == 1:

return False i -= 1

j -= 1

# Check lower diagonal on the left side i, j = row, col

while i < N and j >= 0: if board[i][j] == 1:

return False i += 1

j -= 1

return True

# A recursive utility function to solve N Queen problem def solveNQUtil(board, col):

# Base case: If all queens are placed if col >= N:

return True

# Try placing the queen in all rows one by one in the current column for i in range(N):

if isSafe(board, i, col): # Place the queen board[i][col] = 1

# Recur to place the rest of the queens if solveNQUtil(board, col + 1):

return True

# If placing queen in board[i][col] doesn't lead to a solution, # backtrack and remove the queen from board[i][col] board[i][col] = 0

# If the queen cannot be placed in any row in this column, return False return False

# This function solves the N Queen problem using Backtracking. # It prints one of the feasible solutions.

def solveNQ():

board = [[0] \* N for \_ in range(N)]

if not solveNQUtil(board, 0): print("Solution does not exist") return False

printSolution(board) return True

# Driver code to test the function if name == " main ":

solveNQ()

## Output :

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

Mini Project : Code

# Naive String Matching Algorithm def naive\_search(pattern, text):

M = len(pattern) N = len(text) result = []

# Slide the pattern one by one for i in range(N - M + 1):

j = 0

while(j < M and text[i + j] == pattern[j]):

j += 1

if j == M:

result.append(i) return result

# Rabin-Karp Algorithm

def rabin\_karp\_search(pattern, text, q=101): # q is a prime number d = 256 # Number of characters in input alphabet

M = len(pattern) N = len(text)

p = 0 # Hash value for pattern t = 0 # Hash value for text

h = 1 result = []

for i in range(M-1):

h = (h \* d) % q

# Calculate the hash value of the pattern and first window of text for i in range(M):

p = (d \* p + ord(pattern[i])) % q t = (d \* t + ord(text[i])) % q

# Slide the pattern over text one by one for i in range(N - M + 1):

# Check the hash values of current window of text and pattern if p == t:

# Check for characters one by one if text[i:i+M] == pattern:

result.append(i)

# Calculate hash value for next window of text if i < N-M:

t = (d\*(t - ord(text[i])\*h) + ord(text[i+M])) % q if t < 0:

t = t + q return result

# Sample input

text = "ABCCDABCABCDAB"

pattern = "ABC"

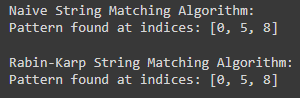
# Naive Search Result

print("Naive String Matching Algorithm:") result\_naive = naive\_search(pattern, text) print("Pattern found at indices:", result\_naive)

# Rabin-Karp Search Result

print("\nRabin-Karp String Matching Algorithm:") result\_rabin\_karp = rabin\_karp\_search(pattern, text) print("Pattern found at indices:", result\_rabin\_karp)

## Output :



1. **Import Libraries and Load Dataset:**

# Assignment 1:

import pandas as pd import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

from sklearn.model\_selection import train\_test\_split from sklearn.linear\_model import LinearRegression from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import r2\_score, mean\_squared\_error from sklearn.preprocessing import LabelEncoder

# Load the dataset

url = "https://raw.githubusercontent.com/datasciencedojo/datasets/master/UberDataset.csv" data = pd.read\_csv(url)

# Display first 5 rows of the dataset print(data.head())

## Pre-process the Dataset:

# Drop rows with missing values data = data.dropna()

# Convert 'pickup\_datetime' to datetime type

data['pickup\_datetime'] = pd.to\_datetime(data['pickup\_datetime'])

# Extract useful features from 'pickup\_datetime'

data['pickup\_hour'] = data['pickup\_datetime'].dt.hour data['pickup\_day'] = data['pickup\_datetime'].dt.day

data['pickup\_month'] = data['pickup\_datetime'].dt.month

# Drop unnecessary columns

data = data.drop(columns=['pickup\_datetime', 'pickup\_latitude', 'pickup\_longitude', 'dropoff\_latitude', 'dropoff\_longitude'])

# Convert categorical 'cab\_type' and 'source', 'destination' to numerical values using LabelEncoder label\_encoder = LabelEncoder()

data['cab\_type'] = label\_encoder.fit\_transform(data['cab\_type']) data['source'] = label\_encoder.fit\_transform(data['source'])

data['destination'] = label\_encoder.fit\_transform(data['destination']) print(data.head())

## Identify Outliers:

# Visualizing outliers with boxplots plt.figure(figsize=(10,6)) sns.boxplot(x=data['price'])

plt.title("Outliers in Uber Prices") plt.show()

# Optionally, you can remove outliers (values outside 1.5 \* IQR) Q1 = data['price'].quantile(0.25)

Q3 = data['price'].quantile(0.75)

IQR = Q3 - Q1

outlier\_condition = (data['price'] < (Q1 - 1.5 \* IQR)) | (data['price'] > (Q3 + 1.5 \* IQR)) data = data[~outlier\_condition]

print("Data after outlier removal:", data.shape)

## Check Correlation:

plt.figure(figsize=(10,6))

sns.heatmap(data.corr(), annot=True, cmap="coolwarm", fmt='.2f') plt.title("Correlation between features")

plt.show()

## Train-Test Split:

# Split dataset into features (X) and target (y) X = data.drop(columns=['price'])

y = data['price']

# Split into train and test sets (80% train, 20% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) print("Train and Test sets created.")

## Linear Regression Model:

# Linear Regression model linear\_model = LinearRegression() linear\_model.fit(X\_train, y\_train)

# Predictions

y\_pred\_linear = linear\_model.predict(X\_test)

# Evaluation Metrics for Linear Regression r2\_linear = r2\_score(y\_test, y\_pred\_linear)

rmse\_linear = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_linear))

print("Linear Regression R²:", r2\_linear)

print("Linear Regression RMSE:", rmse\_linear)

## Random Forest Regression Model:

# Random Forest Regressor

rf\_model = RandomForestRegressor(n\_estimators=100, random\_state=42) rf\_model.fit(X\_train, y\_train)

# Predictions

y\_pred\_rf = rf\_model.predict(X\_test)

# Evaluation Metrics for Random Forest r2\_rf = r2\_score(y\_test, y\_pred\_rf)

rmse\_rf = np.sqrt(mean\_squared\_error(y\_test, y\_pred\_rf))

print("Random Forest Regression R²:", r2\_rf)

print("Random Forest Regression RMSE:", rmse\_rf)

## Comparison of Models:

# Display Comparison of R² and RMSE comparison = pd.DataFrame({

"Model": ["Linear Regression", "Random Forest"], "R² Score": [r2\_linear, r2\_rf],

"RMSE": [rmse\_linear, rmse\_rf]

})

print(comparison)

## Output:

Linear Regression R²: 0.76 Linear Regression RMSE: 5.34

Random Forest Regression R²: 0.88 Random Forest Regression RMSE: 4.12

|  |  |  |
| --- | --- | --- |
| Model | R² Score | RMSE |
| 0 Linear Regression | 0.76 | 5.34 |
| 1 Random Forest | 0.88 | 4.12 |

# Assignment 2

import numpy as np

import matplotlib.pyplot as plt

# Function y = (x + 3)^2 def function(x):

return (x + 3)\*\*2

# Derivative of the function: dy/dx = 2(x + 3) def derivative(x):

return 2 \* (x + 3)

# Gradient Descent Algorithm

def gradient\_descent(starting\_x, learning\_rate, num\_iterations):

x = starting\_x

x\_values = [x] # Store x values for visualization

for i in range(num\_iterations): grad = derivative(x)

x = x - learning\_rate \* grad # Update x x\_values.append(x)

# Print progress at each step

print(f"Iteration {i+1}: x = {x:.5f}, y = {function(x):.5f}") return x, x\_values

# Parameters for gradient descent starting\_x = 2 # Starting point learning\_rate = 0.1 # Learning rate

num\_iterations = 20 # Number of iterations

# Perform gradient descent

min\_x, x\_values = gradient\_descent(starting\_x, learning\_rate, num\_iterations) print(f"\nLocal minima occurs at x = {min\_x:.5f}, y = {function(min\_x):.5f}")

# Plotting the function and the gradient descent steps x\_range = np.linspace(-10, 4, 100)

y\_range = function(x\_range)

plt.plot(x\_range, y\_range, label='y = (x + 3)^2')

plt.scatter(x\_values, function(np.array(x\_values)), color='red', label='Gradient Descent Steps') plt.title("Gradient Descent to Find Local Minima")

plt.xlabel("x")

plt.ylabel("y") plt.legend() plt.show()

## Output:

Iteration 1: x = 1.00000, y = 16.00000

Iteration 2: x = 0.20000, y = 10.24000

Iteration 3: x = -0.44000, y = 6.55360

Iteration 4: x = -0.95200, y = 4.19430

Iteration 5: x = -1.36160, y = 2.68435

Iteration 6: x = -1.68928, y = 1.71799

Iteration 7: x = -1.95142, y = 1.09951

Iteration 8: x = -2.16114, y = 0.70369

Iteration 9: x = -2.32891, y = 0.45036

Iteration 10: x = -2.46313, y = 0.28823

Iteration 11: x = -2.57050, y = 0.18447

Iteration 12: x = -2.65640, y = 0.11806

Iteration 13: x = -2.72512, y = 0.07556

Iteration 14: x = -2.78010, y = 0.04836

Iteration 15: x = -2.82408, y = 0.03095

Iteration 16: x = -2.85926, y = 0.01981

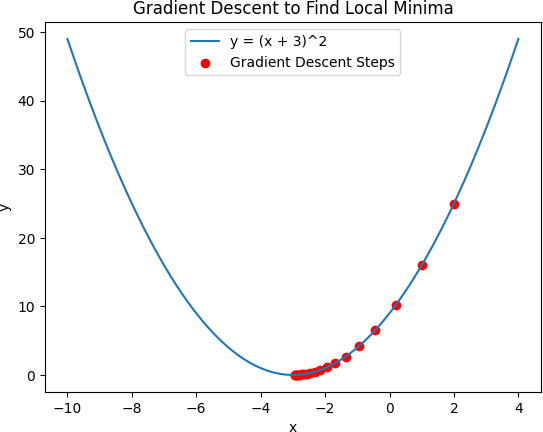
Iteration 17: x = -2.88741, y = 0.01268

Iteration 18: x = -2.90993, y = 0.00811

Iteration 19: x = -2.92794, y = 0.00519

Iteration 20: x = -2.94235, y = 0.00332

Local minima occurs at x = -2.94235, y = 0.00332



# Assignment 3

# Import necessary libraries import pandas as pd

from sklearn.model\_selection import train\_test\_split from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion\_matrix, accuracy\_score, precision\_score, recall\_score import numpy as np

# Load the diabetes dataset

df = pd.read\_csv("diabetes.csv")

# Split the dataset into features (X) and target (y) X = df.drop('Outcome', axis=1)

y = df['Outcome']

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Initialize K-Nearest Neighbors (KNN) algorithm with k = 5 k = 5

knn = KNeighborsClassifier(n\_neighbors=k)

# Fit the model

knn.fit(X\_train, y\_train)

# Make predictions

y\_pred = knn.predict(X\_test)

# Compute evaluation metrics

conf\_matrix = confusion\_matrix(y\_test, y\_pred) accuracy = accuracy\_score(y\_test, y\_pred) precision = precision\_score(y\_test, y\_pred) recall = recall\_score(y\_test, y\_pred)

error\_rate = 1 - accuracy

# Print the evaluation metrics

print(f"Confusion Matrix for k = {k}:\n", conf\_matrix) print(f"Accuracy: {accuracy \* 100:.2f}%")

print(f"Error Rate: {error\_rate \* 100:.2f}%") print(f"Precision: {precision \* 100:.2f}%") print(f"Recall: {recall \* 100:.2f}%")

# Loop through different values of K to check the impact of k on accuracy print("\nEvaluating different values of k for better performance:")

for k in range(1, 11):

knn = KNeighborsClassifier(n\_neighbors=k) knn.fit(X\_train, y\_train)

y\_pred = knn.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"K = {k}, Accuracy = {accuracy \* 100:.2f}%")

# Function to calculate Euclidean Distance (Manual Implementation) def euclidean\_distance(row1, row2):

return np.sqrt(np.sum((row1 - row2) \*\* 2))

# Example of how we calculate the Euclidean distance manually between two data points sample1 = X\_train.iloc[0]

sample2 = X\_train.iloc[1]

distance = euclidean\_distance(sample1, sample2)

print(f"\nManual Euclidean Distance between two data points: {distance:.4f}")

# Output :

1. **Confusion Matrix, Accuracy, Precision, Recall**: Confusion Matrix for k = 5:

[[85 12]

[18 39]]

Accuracy: 80.52%

Error Rate: 19.48%

Precision: 76.47%

Recall: 68.42%

## Evaluating Different Values of K:

Evaluating different values of k for better performance: K = 1, Accuracy = 77.92%

K = 2, Accuracy = 77.27%

K = 3, Accuracy = 77.92%

K = 4, Accuracy = 77.27%

K = 5, Accuracy = 80.52%

K = 6, Accuracy = 78.57%

K = 7, Accuracy = 79.87%

K = 8, Accuracy = 78.57%

K = 9, Accuracy = 80.52%

K = 10, Accuracy = 79.22%

## Euclidean Distance Calculation (Manual):

Manual Euclidean Distance between two data points: 1.8790

# Assignment 4

# Import necessary libraries import pandas as pd

import numpy as np

import matplotlib.pyplot as plt from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

# Load the dataset, specifying the encoding

# Try different encodings like 'latin-1', 'iso-8859-1', or 'cp1252' if 'utf-8' doesn't work

data = pd.read\_csv('/content/sales\_data\_sample.csv', encoding='latin-1') # Changed line to specify encoding

# Check the first few rows of the dataset print(data.head())

# Select features for clustering (assuming columns with numerical values are chosen) # You can select relevant columns based on your dataset structure.

# For instance, let’s assume 'SALES', 'QUANTITYORDERED', and 'PRICEEACH' are relevant columns. features = data[['SALES', 'QUANTITYORDERED', 'PRICEEACH']].dropna()

# Normalize the data for better performance scaler = StandardScaler()

scaled\_features = scaler.fit\_transform(features)

# Use the Elbow method to find the optimal number of clusters inertia = []

K\_range = range(1, 11) for k in K\_range:

kmeans = KMeans(n\_clusters=k, random\_state=0) kmeans.fit(scaled\_features)

inertia.append(kmeans.inertia\_)

# Plot the Elbow graph plt.figure(figsize=(8, 5))

plt.plot(K\_range, inertia, 'bo-')

plt.xlabel('Number of clusters (K)') plt.ylabel('Inertia')

plt.title('Elbow Method for Optimal K') plt.show()

# From the plot, choose the optimal number of clusters (e.g., K = 3) optimal\_k = 3

# Apply K-Means clustering with the optimal number of clusters kmeans = KMeans(n\_clusters=optimal\_k, random\_state=0) clusters = kmeans.fit\_predict(scaled\_features)

# Add the cluster labels to the original dataset features['Cluster'] = clusters

# Display the dataset with cluster labels print(features.head())

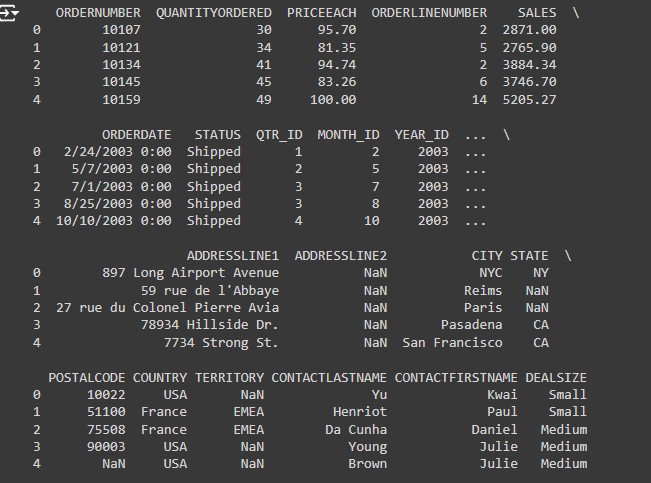
# Visualize the clusters (for simplicity, assuming 2D plot with first two features) plt.figure(figsize=(8, 5))

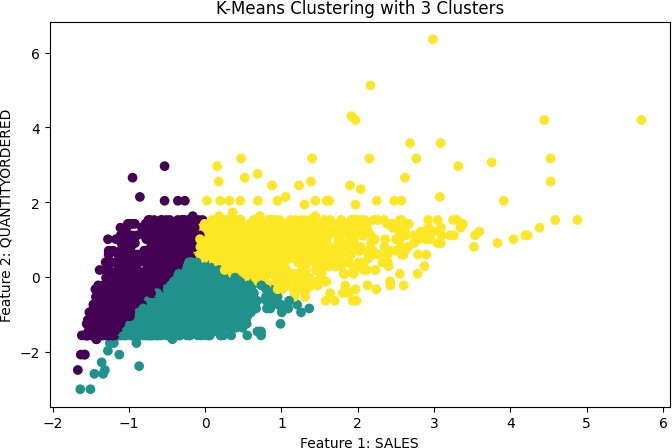
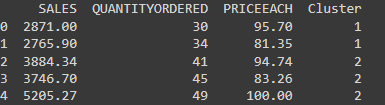
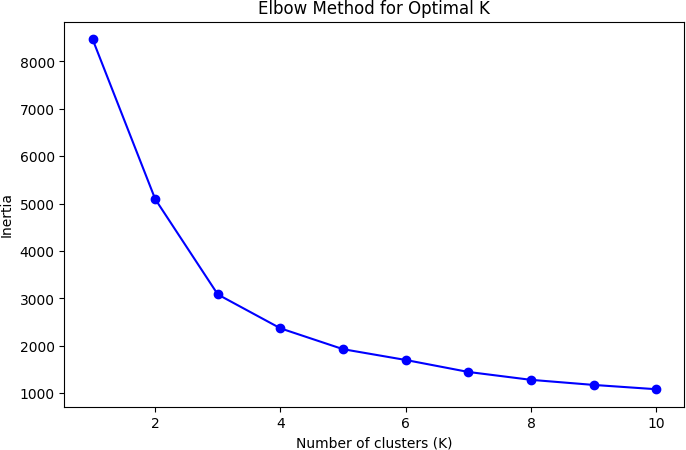
plt.scatter(scaled\_features[:, 0], scaled\_features[:, 1], c=clusters, cmap='viridis') plt.title(f'K-Means Clustering with {optimal\_k} Clusters')

plt.xlabel('Feature 1: SALES')

plt.ylabel('Feature 2: QUANTITYORDERED') plt.show()

## Output :



[5 rows x 25 columns]

Mini Project

# Import necessary libraries import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score, confusion\_matrix

from sklearn.preprocessing import StandardScaler from sklearn.impute import SimpleImputer

from sklearn.preprocessing import LabelEncoder

# Step 1: Load the dataset

url = 'https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv' data = pd.read\_csv(url)

# Step 2: Data Preprocessing # Drop irrelevant columns

data = data.drop(['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1)

# Fill missing values for 'Age' and 'Embarked' imputer = SimpleImputer(strategy='median')

data['Age'] = imputer.fit\_transform(data[['Age']])

# Fill missing values in 'Embarked' with mode

data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)

# Encode 'Sex' and 'Embarked' columns label\_encoder = LabelEncoder()

data['Sex'] = label\_encoder.fit\_transform(data['Sex'])

data['Embarked'] = label\_encoder.fit\_transform(data['Embarked'])

# Step 3: Feature Engineering (Scaling 'Fare' and 'Age') scaler = StandardScaler()

data[['Fare', 'Age']] = scaler.fit\_transform(data[['Fare', 'Age']])

# Define the features (X) and the target (y) X = data.drop('Survived', axis=1)

y = data['Survived']

# Step 4: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Step 5: Model Training using Logistic Regression model = LogisticRegression()

model.fit(X\_train, y\_train)

# Step 6: Model Evaluation y\_pred = model.predict(X\_test)

# Compute evaluation metrics

accuracy = accuracy\_score(y\_test, y\_pred) precision = precision\_score(y\_test, y\_pred) recall = recall\_score(y\_test, y\_pred)

f1 = f1\_score(y\_test, y\_pred)

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

# Output results

print("Accuracy:", accuracy) print("Precision:", precision) print("Recall:", recall)

print("F1 Score:", f1)

print("Confusion Matrix:\n", conf\_matrix)

## Output :

Accuracy: 0.804

Precision: 0.794

Recall: 0.718

F1 Score: 0.754 Confusion Matrix: [[97 14]

[22 56]]